Structure de données pour systèmes temps-réel multiprocesseur : l'exemple des arbres rouge-noirs

Frédéric Fauberteau
Université Paris-Est Marne-la-Vallée
LIGM, UMR CNRS 8049
5, bd Descartes Champs-sur-Marne
77454 Marne-la-Vallée CEDEX 2
frederic.fauberteau@univ-paris-est.fr

Serge Midonnet
Université Paris-Est Marne-la-Vallée
LIGM, UMR CNRS 8049
5, bd Descartes Champs-sur-Marne
77454 Marne-la-Vallée CEDEX 2
serge.midonnet@univ-paris-est.fr

RÉSUMÉ

Dans un système temps réel multitâches, le partage de structures de données impose la synchronisation des opérations de mise à jour et conduit à des accès séquentiels. Ce qui ne permet pas de profiter pleinement des ressources processeur disponibles. Dans cet article, nous présentons un nouveau type de structure de données adapté aux environnements temps réel multiprocesseur. Les données sont stockées dans les feuilles permettant une synchronisation locale. L'équilibrage de la structure n'est plus effectué lors des mises à jour mais dans les temps creux du système. Nous montrons son efficacité en mesurant les temps de réponse moyen et en estimant le comportement pire cas.

Mots clés

Structures de données, concurrence, multi-processeur, temps réel.

1. INTRODUCTION

La spécification RTSJ (*Real-Time Specification for Java*) [6] offre les outils permettant le développement d'applications temps réel avec le langage Java sans limitation quand aux structures proposées par Java. Ces dernières n'offrent toutefois pas le déterminisme nécessaire au temps réel.

Lors de la conception d'une application, le choix du type de structure de données utilisé dépend des propriétés recherchées. L'utilisation d'une table d'association (ou table de hachage) peut s'avérer intéressante pour stocker un gros volume de données avec une contrainte forte sur les temps d'accès. Mais si on ne dispose que d'une très faible capacité mémoire – dans le cadre d'applications embarquées par exemple – et que le volume de données est limité, on préférera utiliser un tableau.

Et bien que bon nombre de structures proposées par Java soient conçues pour une utilisation concurrente, les mécanismes mis en oeuvre se limitent souvent à la pose d'un verrou global sur la structure. Mais des efforts sont fait pour proposer des API plus adaptées aux contraintes temps réel.

Dans [3], Dautelle propose un ensemble de structures pour Java dont le comportement est déterministe. Nous allons présenter une structure de données d'arbre binaire de recherche relâché, l'arbre chromatique, proposée par Guibas et Sedgewick [7] et nous allons montrer que son utilisation est déterministe. Cette dernière est dérivée de l'arbre rouge-noir, structure permettant entre autre d'implanter une table d'association clé-valeur où les données sont triées par ordre de clé croissante. Nous verrons comment une telle structure relâchée peut améliorer les performances d'une application les utilisant dans un contexte multiprocesseur.

Le reste de cet article est organisé comme suit. Dans la Section 2, nous présentons la spécification pour Java temps réel. Dans la Section 3, nous présentons notre conception des structures de données pour le temps réel. Nous expliquons le fonctionnement de l'arbre rouge-noir et nous voyons comment il peut être modifié pour réduire les temps de blocage dans un environnement multiprocesseur. Dans la Section 4, nous commentons les résultats des simulations que nous avons réalisées avec nos implantations de ces structures. Enfin, dans la Section 5, nous concluons en proposant quelques perspectives à ce travail.

2. RTSJ

Les objectifs de la Spécification Java pour le Temps Réel (RTSJ) [6] sont doubles : étendre les APIs de Java pour programmer des applications temps réel et spécifier les caractéristiques d'une machine virtuelle Java Temps Réel (RTJVM). Les solutions retenues portent sur la gestion des priorités et l'ordonnancement des *thread*. De nouveaux modèles de *thread* sont proposés en fonction de leur mode d'activation : synchrones (*RealtimeThread*) et asynchrones (*AsyncEventHandler*). La spécification permet de leur associer des paramètres temporels (coût, période, échéance). Un contrôle de la charge du système peut être effectué à priori par appel à une méthode d'analyse d'ordonnançabilité mais aussi en ligne par le jeu de détecteurs de fautes et des *handler* de traitement associés. On pourra par exemple détecter des dépassements de coût ou le non respect de la périodicité minimale d'un *thread* sporadique ainsi que le dépassement d'une échéance. Second élément important, les moniteurs utilisés pour les synchronisations doivent implanter un algorithme

qui borne les inversions de priorité [14]. Enfin RTSJ revisite la gestion de la mémoire Java et divise la mémoire en régions. Le tas géré par le *garbage collector* du système qu'il soit temps réel ou pas. La mémoire *scope* qui peut être détruite si et seulement si aucune entité ordonnançable ne lui est associée. La mémoire immortelle pour laquelle les objets asociés ont la même durée de vie que l'application. L'utilisation d'un *garbage collector* temps réel n'est pas obligatoire, toutefois de nombreux travaux comme ceux décrit dans [12] ont conduit à l'intégration de ce type d'objets dans plusieurs RTJVM (Perc,JamaicaVM). Pour aller plus loin dans la compréhension de cette spécification, les lecteurs pourront se référer aux ouvrages [13] et [4]. Plus récemment un nouvel appel à spécification (JSR-282) a été publiée et ouvre la spécification aux environnements multiprocesseurs.

3. STRUCTURE DE DONNÉES POUR LE TEMPS RÉEL

Dans cette section nous présentons la structure d'arbre rouge-noir dans un premier temps. Nous décrivons ensuite les modifications apportées à cette structure pour obtenir un arbre chromatique. Enfin, nous voyons pourquoi cette structure est plus performante dans un environnement multiprocesseur. Avant de présenter ces structures de données, nous définissons ce qui pour nous caractérise une structure de données temps réel.

DÉFINITION 1. *Une structure de donnée temps réel est une structure de donnée :*
- *dont le comportement est déterministe ;*
- *pour laquelle on dispose des outils nécessaires au calcul de son coût d'exécution pire cas.*

3.1 Coût d'exécution pire cas (WCET)

L'estimation des durées pire cas d'exécution est un problème complexe. La sous estimation des cette durée pire cas peut conduire à des dépassements (fautes temporelles) lors de l'exécution du système voire des défaillances dans certains cas (non respect de l'échéance). Il existe deux manières d'estimer la durée d'exécution pire cas (WCET) d'une tâche [8] . L'une repose sur l'analyse dite statique du code, méthode très complexe et non portable car les instructions analysées dépendent de l'architecture sous jacente (binaire). Dans le cas de systèmes Java, le binaire ne sera produit que lors de l'exécution. Dans certains cas, l'analyse du *bytecode* est toutefois possible comme dans certains cas où le processeur éxécute en natif du code Java (JOP) [11] . Nous utilisons une approche plus empirique dite dynamique qui consiste à effectuer un ensemble d'exécutions et à extraire de ces mesures la pire valeur.

3.2 Arbre rouge-noir

La structure d'arbre rouge-noir est connue depuis longtemps et a été présentée pour la première fois par Bayer *et al.* [1]. Il s'agit d'un arbre binaire de recherche. Un arbre est une structure composée de noeuds reliés entre eux par des arrêtes et qui est dépourvu de cycle. Un arbre binaire est un arbre où chaque noeud peut avoir au plus 2 noeuds fils. Un arbre binaire de recherche est un arbre binaire où chaque noeud n est soumis aux contraintes suivantes :
- tout noeud n doit posséder une donnée servant de clé ;
- tout noeud appartenant au sous-arbre gauche de n doit avoir une clé dont la valeur est inférieure où égale à la valeur de la clé de n ;
- tout noeud appartenant au sous-arbre droit de n doit avoir une clé dont la valeur est supérieure à la valeur de la clé de n.

Dans le cas d'un arbre binaire de recherche équilibré, c'est-à-dire lorsque la différence des hauteurs de deux sous-arbres n'excède pas 1, les opérations de recherche et de mise à jour prennent un temps $O(log(N))$ où N est le nombre de noeuds dans l'arbre. Un arbre rouge-noir est un arbre binaire de recherche qui a la propriété de s'équilibrer après chaque opération de mise à jour pour pouvoir garantir ce temps $O(log(N))$. Les noeuds de l'arbre contiennent une valeur binaire (rouge ou noir) en plus de la clé qui permet de renseigner sur l'état de l'arbre. Après une insertion ou une suppression au niveau du noeud n_i, la couleur de n_i est mise à jour, et cette mise à jour peut se répercuter jusqu'à la racine. Dans le cas où un déséquilibre est détecté, en fonction de la couleur des noeuds, un balancement est opéré pour rétablir l'équilibre de l'arbre. Bien que les opérations de maintenance d'un arbre rouge-noir puissent paraître compliquées, cette structure offre de bonnes performances et est toujours utilisée. En effet, l'API de Java fournit une structure de données, *TreeMap*, qui est une table d'association où les clés sont triées suivant un ordre donné. Très récemment, un nouvel ordonnanceur de processus a été ajouté au noyau Linux, le *Completly Fair Scheduler*, qui utilise comme structure de donnée sous-jacente un arbre rouge-noir [10, 9].

Au regard de la DÉFINITION 1, l'arbre rouge-noir ne nous apparaît pas comme étant une structure temps réel. En effet, il est très difficile de prévoir en fonction de l'opération de mise à jour si un équilibrage sera fait et quel serait le coût de cet équilibrage. Il reste tout de même la possibité de calculer un coût d'exécution où on considère que l'arbre est toujours dans le pire état de déséquilibre. Une opération de mise à jour (insertion ou suppression), dans le pire cas, peut induire un rebalancement complet de l'arbre. Cela signifie qu'une tâche voulant effectuer une mise à jour ou simplement une recherche dans l'arbre devra attendre qu'aucune autre tâche n'utilise l'arbre. Il faut donc mettre en place un mécanisme de verrouillage global sur l'ensemble de la structure. Plus le nombre de tâche voulant avoir accès à l'arbre est grand, et plus le temps de blocage sera important. Ce phénomène peut avoir des conséquences dramatiques s'il s'agit d'une application temps réel car certains protocoles de synchronisation échangent les priorités des tâches pour éviter les interblocages. Une tâche de forte priorité peut ainsi être bloquée par une tâche de priorité plus faible pendant un temps très long.

3.3 Arbre chromatique

Les arbres chromatiques sont aussi des arbres binaires de recherche. À la différence des arbres rouge-noir, ils ne sont pas automatiquement équilibrés. De plus, les noeuds ne contiennent plus une valeur binaire (rouge ou noir) pour spécifier leur structure interne, mais une valeur entière spécifiant un poids. Ils ont été proposés en réponse aux inconvénients des arbres rouge-noir exposés à la fin de la sous-section précédente. Dans un arbre chromatique, les données sont stockées dans les feuilles – les noeuds qui n'ont pas fils – et les noeuds internes servent uniquement à guider la recherche. Ainsi, une

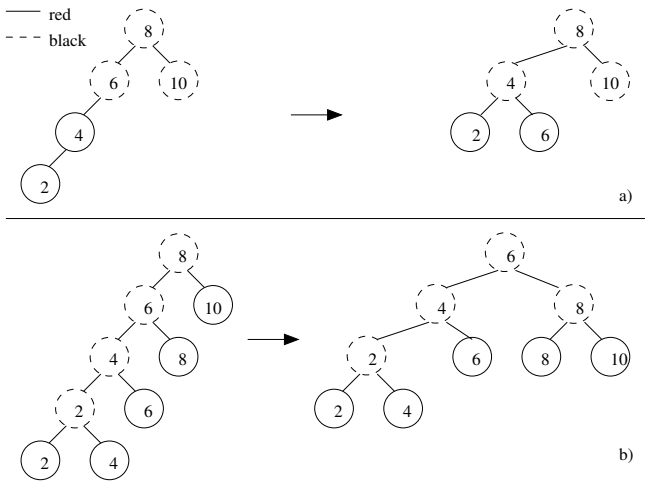

Figure 1: Équilibrage a) d'une arbre rouge-noir, b) d'un arbre chromatique

tâche peut insérer ou supprimer une donnée dans l'arbre sans en changer la structure interne. Lorsqu'une donnée doit être insérée (ou supprimée) et que la recherche de l'emplacement de cette donnée s'est terminée dans le noeud n_f, un verrou en écriture doit être posé sur n_f pour empêcher tout autre modification de ce noeud. Plusieurs tâches peuvent ainsi faire des recherches dans l'arbre pendant qu'une autre tâche faire une mise à jour de l'arbre. Ce comportement est rendu possible car aucun équilibrage n'est effectué après une mise à jour et l'arbre reste donc dans un état stable. Nous pouvons tout de même remarquer que dans le pire cas, à cause de l'absence d'équilibrage, l'arbre chromatique peut devenir une liste. Cela se produit notamment lorsque toutes les données sont insérées de manière ordonnée (suivant l'ordre des clés). Pour pallier à ce phénomène, un algorithme a été proposé pour équilibrer un arbre chromatique après plusieurs opérations de mise à jour succcessives. Cet algorithme nécessite le verrouillage complet de l'arbre. Il consiste à parcourir l'arbre et à effectuer des rotations lorsque les poids des noeuds ont des valeurs particulières. Il faut donc procéder à cet équilibrage pendant qu'aucune tâche n'utilise la structure. C'est pourquoi nous avons proposé d'effectuer cet équilibrage dans les temps creux du système, c'est-à-dire lorsqu'aucune tâche de l'application n'est active [5]. Enfin, pour pouvoir estimer le coût d'exécution pire cas d'utilisation d'un arbre chromatique, il faut pouvoir estimer le coût pire cas de l'équilibrage. Heureusement, le nombre d'opérations de l'algorithme est borné en fonction du nombre de mises à jour effectuées dans l'arbre.

THÉORÈME 1. *(Boyar et Larsen [2]) Soit un arbre contenant initialement $|T|$ noeuds. Considérons que k insertions et s suppressions ont été faites dans l'arbre. Alors le nombre de rebalancements nécessaires pour équilibrer l'arbre est borné par $(k+s)(\lfloor log_2(N+1)\rfloor - 1) - s$, où $N = |T| + 2k$.*

Dans la Figure 1, nous donnons un exemple d'arbre rouge-noir et chromatique.

4. RÉSULTATS DE SIMULATION

Pour nos premières simulations, nous avons implanté une structure respectant l'interface de la collection *TreeMap*, mais utilisant un arbre chromatique à la place d'un arbre rouge-noir (disponible à l'adresse *http ://igm.univ-mlv.fr/AlgoTR/codes/javartstruct.tar.bz2*) . Nous l'avons comparé avec la collection *TreeMap* originale sur une architecture monoprocesseur. Nous montrons dans la Figure 2 que le coût des insertions dans le pire cas, c'est-à-dire des données dont les clés sont triées. Nous voyons que le coût de l'insertion dans l'arbre chromatique est légèrement plus faible mais qu'il tend à se confondre avec celui de l'insertion dans un arbre rouge-noir. Dans cette simulation, nous n'avons pas équilibré l'arbre chromatique. C'est pourquoi les performances se dégradent si vite. En effet, là où le parcours de l'arbre nécessite $log(N)$ opérations de comparaison de clé dans l'arbre rouge-noir, il en nécessite N dans l'arbre chromatique. Bien que le coût d'une opération de mise à jour dans l'arbre chromatique soit allégé du coût de l'équilibrage, la hauteur de l'arbre peut augmenter plus vite que celle d'un arbre rouge-noir. Il faut donc mettre en place un mécanisme d'équilibrage lorsque l'arbre chromatique est trop déséquilibré.

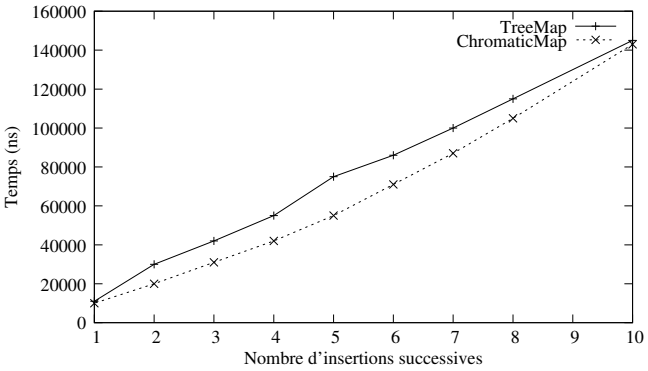

Figure 2: Comparaison des coûts d'insertion pire cas entre un arbre rouge-noir et un arbre chromatique

Nous avons fait d'autres test d'exécution sur une architecture multiprocesseur / multicoeur. Comme nous ne disposons pas encore de machine virtuelle Java respectant les contraintes nécessaires pour faire du temps réel multiprocesseur, nous avons implanté une version en C/POSIX des arbres rouge-noir et chromatiques pour pouvoir profiter des mécanismes de priorité et de synchronisation temps réel (disponible à l'adresse *http ://igm.univ-mlv.fr/AlgoTR/codes/crtstruct.tar*
Pour éviter les problèmes de mesure liés à l'allocation dynamique de mémoire, nous avons implanté pour ces structures un allocateur mémoire en temps constant. Nous avons exécuter la simulation sur une machine disposant de 2 processeurs incluant chacun 4 coeurs. Nous avons fait varier le nombre de (*thread*) de 2 à 64 pour pouvoir évaluer les performances des structures dans un environnement fortement concurrent. Les données insérées sont générés aléatoirement. Nous montrons donc dans la Figure 3 le coût d'insertion moyen. Encore une fois, nous n'avons pas procédé à l'équilibrage de l'arbre chromatique. Nous pouvons constater que dans le cas de l'arbre rouge-noir, le coût des opérations est fortement impacté par

les temps de blocage.

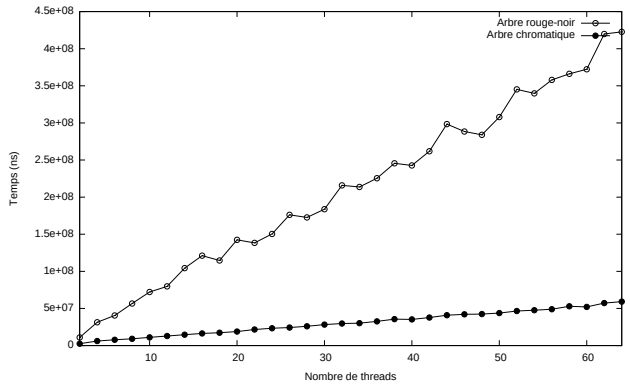

Figure 3: Comparaison des coûts d'insertion en moyenne entre un arbre rouge-noir et un arbre chromatique

5. CONCLUSION ET DÉVELOPPEMENT FUTUR

Nous avons présenté une structure de données utilisée dans des applications réelles (*TreeMap* ou ordonnanceur *CFS* de Linux). Nous avons montré que cette structure n'était pas forcément adaptée pour une utilisation dans une application temps réel devant s'exécuter sur une architecture multiprocesseur. Nous avons donc présenté une structure modifiée et montré en quoi elle est mieux adaptée pour l'estimation du coût d'utilisation dans le pire cas. Nous avons aussi présenté nos résultats de simulation basées sur des implantations en C et en Java. Notre objectif est maintenant de proposer d'autres structures modifiées permettant d'améliorer des structures existantes.

6. RÉFÉRENCES

[1] R. Bayer and E. M. Mccreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1 :173–189, 1972.

[2] J. Boyar and K. S. Larsen. Efficient rebalancing of chromatic search trees. *J. Comput. Syst. Sci.*, 49(3) :667–682, 1994.

[3] J.-M. Dautelle. Fully time deterministic java. In *AIAA Space 2007*, 2007.

[4] P. Dibble. *Real-Time Java Platform Programming : Second Edition*. BookSurge Publishing, 2008.

[5] F. Fauberteau and S. Midonnet. Worst case analysis of TreeMap data structure. In *Proceedings of the 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC'08)*, pages 33–36, Rennes, France, Oct. 2008. Short Paper.

[6] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[7] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.

[8] R. Kirner and P. P. Puschner. Classification of wcet analysis techniques. In *ISORC*, pages 190–199, 2005.

[9] A. Kumar. Multiprocessing with the completely fair scheduler. http ://www.ibm.com/developerworks/linux/library/l-cfs/ ?ca=dgr-lnxw06CFC4Linux.

[10] I. Molnar. Modular scheduler core and completely fair scheduler [cfs]. http ://lwn.net/Articles/230501/.

[11] M. Schoeberl and R. Pedersen. Wcet analysis for a java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 201–211. ACM, 2006.

[12] F. Siebert. *Hard Real-Time Garbage Collection in Modern Object Oriented Programming Languages*. Books on Demand GmbH, 2002.

[13] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[14] A. J. Wellings, A. Burns, O. M. dos Santos, and B. M. Brosgol. Integrating priority inheritance algorithms in the real-time specification for java. In *ISORC*, pages 115–123, 2007.